International Journal of Engineering Education Vol. 32, No. 3(A), pp. 1069–1077, 2016
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2016 TEMPUS Publications.

Fine-Grained Recording of Student Programming Sessions to Improve Teaching and Time Estimations*

DANIEL TOLL and TOBIAS OLSSON
Linnaeus University, Nygatan 18 B, 392 34 Kalmar, Sweden. E-mail: daniel.toll@lnu.se, tobias.ohlsson@lnu.se

MORGAN ERICSSON and ANNA WINGKVIST
Linnaeus University, Universitetsplatsen 1, 351 95 Växjö, Sweden. E-mail: morgan.ericsson@lnu.se, anna.wingkvist@lnu.se

It is not possible to directly observe how students work in an online programming course. This makes it harder for teachers to help struggling students. By using an online programming environment, we have the opportunity to record what the students actually do to solve an assignment. These recordings can be analyzed to provide teachers with valuable information. We developed such an online programming tool with fine-grained event logging and used it to observe how our students solve problems. Our tool provides descriptive statistics and accurate replays of a student's programming sessions, including mouse movements. We used the tool in a course and collected 1028 detailed recordings. In this article, we compare fine-grained logging to existing coarse-grained logging solutions to estimate assignment-solving time. We find that time aggregations are improved by including time for active reading and navigation, both enabled by the increased granularity. We also divide the time users spent into editing (on average 14.8%), active use (on average 37.8%), passive use (on average 29.0%), and estimate time used for breaks (on average 18.2%). There is a correlation between assignment solving time for students who pass assignments early and students that pass later but also a case where the times differ significantly. Our tool can help improve computer engineering education by providing insights into how students solve programming assignments and thus enable teachers to target their teaching and/or improve instructions and assignments.

Keywords: computer science education; learning analytics; educational data mining; computer engineering education

1. Introduction

In the past, programming assignments often demanded access to hardware (computers) and software (compilers and tools) that was not easily available outside campus. Students were scheduled to computer laboratory rooms, where participation was often mandatory. This allowed teachers to get a sense for what problems students experienced. Teaching was agile; the class could be gathered during the laboratory session for impromptu lectures that provided more information on a certain topic or a quirk in the development environment. Computers and development tools are now widely available and it is common for students to have a fully working development environment at home; as courses move off-campus and online it is almost mandatory. The scheduled laboratory work is more and more used for examinations or to assist with specific questions handled by teaching assistants. The opportunity for teachers to observe students while programming has almost disappeared.

Based on experience from mixed mode (online and campus students) programming courses, we feel that we do not understand the problems our students face. This feeling is confirmed by for example a survey on student programming behavior [1, page 296]: "This disparity between student and faculty perceptions is an indication of the profound lack of understanding of student software development practices." To understand students, teachers of programming courses have recorded student programming sessions on different granularity levels, from the coarsest level of student assignment submissions to fine level logging of each keystroke [2–7].

We make the following contributions: we implemented an even more fine-grained logging of student programming sessions that includes mouse movements, text selections, and text cursor positions and places an upper limit on time students spend in the editor. We compare these finer-grained logs with previous more coarse-grained in activity aggregations and find that the time for editing is only a fraction of the total amount of time students spend on an assignment. We divide the time spent into activities and use the fine-grained time measurements to determine how the sessions (e.g., problems experienced) of students that finish early compare to students that finishes later.

- **RQ1**. How much of total tool usage time can be estimated using coarse-grained models?
- **RQ2**. What is the fraction of assignment solving time spent on editing, reading and browsing assignment code and how much time do student spend outside of the tool?
- **RQ3**. Can time measurements of students who

* Accepted 12 February 2016.

complete assignments early predict what assignments the main group of students is going to struggle with?

2. Background

In off-campus courses, for example MOOCs, an online programming environment is attractive in part because it provides the opportunity to record events when the students work on their assignments. Most current studies collect data at a coarse-grained level (e.g. submission times, code-changes, or compiler errors) to allow a quantitative data analysis to understand students. BlueJ[1] is a popular free Java development environment aimed at beginners. Jadud [2] used BlueJ to record and investigate the Edit-Compile cycle of novice Java programmers by taking snapshots of the code at each compilation. His findings are in line with other research [1, 5]: a majority of errors are cause by a small subset of compiler errors—most students make the same mistakes. Jadud [2] also studied compilation patterns of students and found that a large portion of students recompile without making any changes to the code. He used the collected compilation error data to define the Error Quotient (EQ), a model to help identify students that struggle. Utting et al. [8] plan to instrument BlueJ to allow for large-scale anonymous data collection and make the collected data available to other researchers. The goal is to allow for more power in quantitative analysis and also to study situations that do not occur very often and thus smaller samples may miss entirely. Helminen et al. [5] include an interactive Python console in their online programming tool. They collect testing behavior statistics on how many students write their own tests or use the assignment's provided tests. Execution error statistics that indicate students' lack of API or language knowledge are also collected. Their granularity level is on recording code submissions, code edits, console interactions and also when the student started their tool and closed it. Vihavainen et al. [7] instrumented NetBeans to collect fine-grained recordings of keystrokes and events, and compare different sampling granularities and report compilation success rates for beginner programmers. Helminen et al. [5] provide an overview of how recordings on various abstraction levels can be used to analyze student behavior. In a number of studies time is estimated using aggregation of coarse-grained events [3, 6, 9]. We have observed students on campus that spend a lot of their programming time thinking and reading before acting. Time measurements that do not include reading time would not do them justice.

[1] http://www.bluej.org/

2.1 Computer science quiz

We developed the Computer Science Quiz tool (CSQUIZ) to bridge the gap between the students' and the teachers' perception of programming assignments. CSQUIZ is a complete programming and learning environment that presents instructions and theory relevant to an assignment. It includes a multi-file code-editor that can execute code, provide feedback from the interpreter, and run automated tests. CSQUIZ is implemented as a web application.

CSQUIZ automatically assesses and records students when they work on their programming assignments. The overall goal is to support teachers with visualizations and descriptive statistics to help find and address shortcomings in the course during its delivery and to perform a detailed analysis and suggest improvements for the next iteration. CSQUIZ provides an exact replay of the students' interaction with the programming environment. The detailed recordings can be played back at different speeds and paused; mouse pointer, text selections, and file changes are all replayed accurately. This allows the teachers to build a deep and detailed knowledge about how students solved a problem.

CSQUIZ records student activities on a fine-grained level. A recorded programming session is a collection of time stamped events. Each session is stored on the server and is identified by a unique hash sum for the student and the name of the assignment. CSQUIZ records the following events:

- CSQUIZ is started, closed or reloaded.
- CSQUIZ becomes active or inactive, for example the browser becomes in focus or out of focus.
- A source file is changed, for example code is written or deleted.
- The text cursor moves.
- Text in editor is selected or deselected.
- A source code file is reset (all changes dropped).
- The produced code is executed (output and errors are logged).
- Switching between source code files or instructions.
- The mouse moves over the editor, recorded at 11 samples per second.
- The mouse buttons are clicked (added in 2015).

We designed and implemented CSQUIZ to record detailed time measurements, for example how long a student spends reading instructions before starting to work on an assignment. Therefore, the instructions and programming area fades if a student is inactive for more than fifteen seconds. This effectively allows us to put an upper limit on the amount of time spent in the tool.

3. Empirical study

We use data collected by CSQUIZ to answer our research questions. Other studies [3, 6, 9] present time aggregated on a coarser granularity, e.g., using compilations or key events. When we base time estimations on coarser granularities we might overestimate the time it takes to complete an assignment by including time that students spend on other activities, or underestimate the time by considering breaks as inactivity and not pauses. Some of our assignments contain around 800 lines of text, and include several files and classes. Reading and browsing has been shown by [10] to be a large part of a programming effort. CQUIZ measures the time spent browsing and reading to capture a larger proportion of the entire student effort. It is thus interesting to compare coarse measurements with fine-grained (RQ1) to determine if there is a difference. To find the fraction of programming time spent on different activities we do temporal aggregation from a sequence of discrete events, where a stream of key press events closely related in time is aggregated into a single measurement of editing time, for example.

To extract several different types of activities, we aggregate on four granularity levels similar to those found in other studies, from the coarsest (G1), reacting on application start up, saves, and compiles, to the finest (G4) based on text visibility. Each granularity level includes the events from coarser levels, e.g., G2 includes all events in G1. This enables us to derive the time for an activity by subtracting one granularity from another. We use an event threshold to determine whether events are connected and should be time aggregated or not. An event that is followed by another event within the threshold is counted as the time between the events. For example, if we use a threshold of ten seconds, two text changes that happen with nine seconds apart would count as one period of nine seconds. With a threshold of three seconds these events would be considered separate. In our aggregation model, a separate event that is not followed by another event within the threshold is counted as 0.5 seconds.

We use CSQUIZ recordings from 66 second-year university students completing 17 different programming assignments. All students did not finish all assignments before the deadline, so we collected a total of 1028 programming sessions. The assignments were divided into two blocks. A first block of ten assignments was introduced on September 1, 2014. During this block, the tool and assignments were updated on two occasions. A minor change in the output visualization was introduced on September 3, 2014 to make it clear to the students if the application did not produce any output. The second update introduced seven new assignments in block two on September 19, 2014. The data collection ended on October 13, 2014.

We collected a second data set during autumn 2015 with a larger group of 83 students. A number of differences exist between the recording conditions of the first and second data sets: the 2015 course material was in English while the 2014 material was in Swedish and some of the programming assignments were changed, replaced, and new assignments were introduced. This second data set consists of 1573 recorded programming sessions that were recorded between August 8, 2015 and October 2, 2015. The larger student group in 2015 consisted of a mix of second and third year students.

3.1 RQ1, Comparing granularities

Each increased granularity level may introduce additional issues. For example, to transition from only recording submissions to also record compilations requires some type of instrumentation of the programming tool. CSQUIZ records events at high frequencies (e.g., mouse and key events) which results in a lot of network traffic and large log files. It also introduces inconveniences for the students by fading out text on inactivity. By increasing the event threshold, events that appear further apart are then regarded as connected. The total time used in the tool could be estimated with a lower sampling granularity level and a higher threshold. How much of total tool usage time can be estimated using coarse-grained models?

3.1.1 Method

We measure on four granularity levels (G1–G4) and use five different thresholds. We use the thresholds three seconds, ten seconds, one minute, five minutes, and fifteen minutes to be able to compare to related works and to reach overestimation for each granularity. Each combination of granularity and threshold forms an aggregation model and each of the 1028 programming sessions from 2014 is aggregated using the resulting 20 different aggregation models. We compare the time aggregation values with CSQUIZ finest granularity (G4) of time students spend in the tool. We compute the baseline Time in Tool (*TiT*) with three-second threshold and compute the error (percentage) for all other models. The value presented is calculated as $(TiT\text{-}GX(Y))/TiT$, where X is the granularity level and Y is the threshold. We present the error percentage and standard deviation to determine the precision of the coarser granularities. A negative percentage indicates an underestimation and a positive indicates an overestimation.

3.1.2 Results

Table 1 presents error averages in percent for total activity Time in Tool depending on sampling granularities and thresholds. The values should be interpreted as follows: a measurement on "Compilations" granularity (G1) and with one minute threshold results in an underestimation of the Time in Tool and only captures on average 22% (–78% error) of the Time in Tool. If we choose a coarser sampling granularity we get either an average underestimation or an overestimation, or get a large standard deviation. For example, measuring text changes (G2) with a threshold of five minutes results in quite accurate readings on average (–2%) but a high standard deviation. Using the "Active Use" granularity (G3) improves the estimations, and we get quite accurate readings with a threshold of one minute. This is not surprising since the students use mouse movement to keep the text visible. If application start up is not included, the Compilation granularity (G1) constantly underestimates the time spent even with large thresholds. At a three-hour threshold it creates acceptable average time but large individual errors.

3.1.3 Discussion

The Active Use (G3) granularity estimations are probably higher than they would be if CSQUIZ did not fade text on inactivity. When looking at playbacks of recordings we sometimes observe a passive period followed by a jerky mouse movement that is used to regain text visibility. The G1 granularity is perhaps unfairly represented here due to the way we treat separate events (as 0.5 seconds), only compilations that are within a threshold of each other are going to matter. Using larger thresholds and finer granularities quickly result in overestimations and capture usage outside of the tool, which we use to answer the next question. It is clear from the values in Table 1 that in order to measure Time in Tool, granularity and threshold matters. We consider capturing mouse movements worth the effort to get accurate readings. Fading text might be too intrusive to the students.

3.2 RQ2, Time spent reading

It is difficult to estimate the reading time without eye-tracking since reading does not necessarily involve any other interaction. However, by fading text on inactivity CSQUIZ provides a definite time cap on tool usage, since students must stay active to be able to continue. It also provides us with an opportunity to measure the time when students move the mouse over the application (active use) and the amount of time the text is visible (passive use). The true time for reading and navigation may be as high as the sum of both but not higher. What fraction of assignment solving time is spent on editing, reading and browsing assignment code and how much time do student spend outside of the tool?

3.2.1 Method

We use the time aggregation model described earlier and divide the time spent into activities. We start by computing a Total Time (*TT*) for all activities linked to an assignment. We measure on G4 but with a fifteen-minute threshold, so we will capture time spent outside of the tool, for example online searches or noise from other sources such as social media. We consider fifteen minutes to be the longest break one would take and still be considered working in one session. We also measure the Time in Tool by measuring on G4 but with a threshold of three seconds. The Editing Time (*ET*) is measured using G2 with a three second threshold and the Active Use Time (*AUT*) is measured on G3 with a three seconds threshold.

We compute the average fraction time for Editing $EF = ET / TT$, Active Use $AU = (AUT - ET) / TT$, time Out of Tool $OT = (TT - TiT) / TT$ and Passive Use $PU = 1 - (EF + AU + OT)$. These values are computed for each recorded programming session and statistics are computed and presented. We calculate these from the recordings from 2014 and 2015. The results are presented separately to determine if the changes to students, teaching and assignment might affect the reading time.

3.2.2 Results

We estimate the time a student spent on reading and

Table 1. Comparing combination of thresholds and time aggregation granularities using 1028 recorded programming sessions to estimate how much of the "Time in Tool" a combination captures on average. A negative value is an underestimation and a positive value an overestimation (*: Baseline Time in Tool)

	3 seconds		**10 seconds**		**1 minute**		**5 minutes**		**15 minutes**	
	Error	**stdev**	**Error**	**stdev**	**Error**	**stdev**	**Error**	**stdev**	**Error**	**stdev**
G1. Compilations	–99%	0.8	–98%	3.4	–78%	22.3	–27%	38.0	+6%	35.7
G2. Text change	–83%	9.3	–73%	13.6	–43%	23.0	–2%	24.5	+17%	31.1
G3. Active Use	–37%	15.7	–21%	13.5	+2%	6.7	+19%	27.5	+32%	59.8
G4. Time in Tool	0% *	0	+1%	1.9	+7%	8.5	+21%	30.3	+35%	72.3

navigating, and present the results from 2014 in Table 2. Compared to Table 1, we show results as percentages of total assignment solving time, including time spent outside the tool. We find that students are actively using the mouse, doing text selections or moving the text cursor 37.9% of the time on average. This time can be further examined using CSQUIZ's replay functionality. The Passive Use time represents time that we cannot account for; we know that the text has not yet faded, but we have no events from the user. When we watch the recordings, the mouse is sometimes moved outside the application, or just stops moving for a while. The amount of such breaks varies a lot between students, perhaps due to equipment. For example, a touchpad may have less frequent input than a mouse. We also find that 18.2% of the time is spent outside the tool on average. These breaks can be up to fifteen minutes long (since we used a fifteen-minute threshold). Longer breaks are not included, since we find it unlikely that students would be actively searching for information on a task and not stay in the tool for longer than fifteen minutes. We have not found any evidence of students trying to solve the tasks in other tools. For short assignments, the time Out of Tool is smaller, and for the longer assignments, students are taking more short breaks, perhaps to search for information or other things. The time Out of Tool median was 11.7% for all assignments.

We present the values from 2015 in Table 3. The fraction of time spent editing time is very close to that of 2014, while Active Use has increased to 40.4% and Passive Use has decreased compared to 2014. The fractions vary depending on assignment as can be seen in Fig. 1.

3.2.3 Discussion

The actual time for reading and navigation is a combination of active (AU) and passive use (PU) but not higher than their sum, on average 66.9% in 2014 and 66.3% in 2015. The reading time could be

Table 2. Estimate time spent on activities based on 1028 recordings in 2014 with 17 different programming assignments. Total time includes breaks up to fifteen minutes

Activity	Average	Median	Min	Max	stdev
Editing (*EF*)	14.8%	12.7%	0.2%	57.0%	9.8
Active Use (*AU*)	37.9%	36.5%	2.8%	96.8%	16.2
Passive Use (*PU*)	29.0%	28.2%	0.0%	75.0%	13.1
Out of Tool (*OT*)	18.2%	11.7%	0.0%	93.2%	19.3
Total Time (*TT*)	100%				

Table 3. Estimate time spent on activities based on 1573 recordings in 2015 with a new student group. Programming assignments and course content were changed from 2014

Activity	Average	Median	Min	Max	stdev
Editing (*EF*)	14.7%	12.6%	0.2%	74.6%	10.0
Active Use (*AU*)	40.4%	39.5%	1.4%	95.5%	16.9
Passive Use (*PU*)	25.9%	24.1%	0.0%	74.1%	11.9
Out of Tool (*OT*)	19.0%	11.3%	0.0%	93.5%	20.9
Total Time (*TT*)	100%				

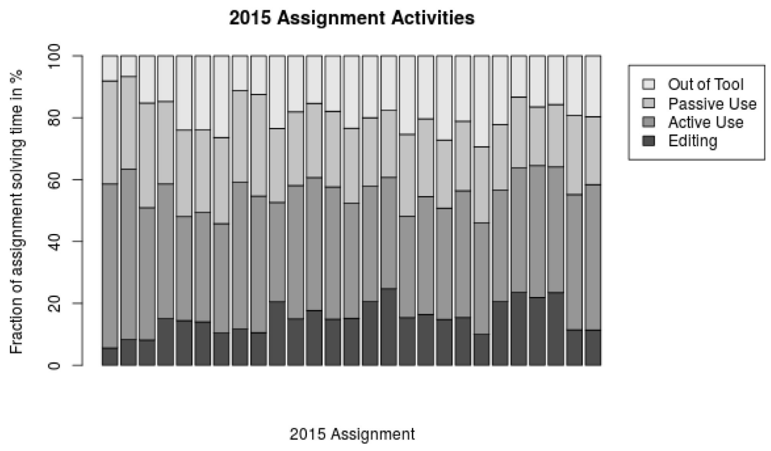

Fig. 1. Fractions of assignment solving time from 1573 recorded programming sessions in 2015. Each column is a programming assignment. The reading time (Active Use + Passive Use) is on average 66.3% but varies among the different assignments.

lower since the student could be moving the mouse over the application but not actually reading. If more control could be added to the measurement situation, for example by tracking eye movements or capturing the entire screen we could improve these values. Such recordings would explain more of what the students do during their short breaks, and provide clues on what happens during the passive time in our tool. If we compare the values in Tables 2 and 3, we find that the students in 2015 on average spends almost the same fraction of time editing and out of tool, but are a bit more active. This might be due to some of the changes done to the course or to the assignments. As shown in Fig. 1, the fraction spent on different activities varies depending on assignment, but reading time is always a large part of the programming effort as can be seen in Fig. 1. The first assignment requires only a small change and thus the editing time was only 5.6%.

3.2.4 RQ3, Predictions

During the autumn of 2014 we used CSQUIZ to focus our teaching resources on problematic assignments. Based on the gathered statistics, we tried to predict which assignments students got most frustrated with. After the course ended, we compared the times to solve from the students that completed the assignments during the first day to those from students that completed the assignments later. We expected three types of effects: (1) motivated students have been reported to start early and this has been seen to correlate with better grades [1], (2) we know from experience that students help each other, and (3) lecture content and more time to study may also effects the results. A good prediction model would allow us to identify what assignments we should spend teaching resources on. For example, we could use lecture time to address issues in a problematic assignment. Can time measurements of students who complete assignments early predict what assignments the main group of students is going to struggle with?

3.2.5 Method

We compare the group of students who completed assignments before September 2, 8AM 2014 (the morning of day two, block one), with students that started and completed the assignments later. We remove recordings of students that started before but did not complete the assignment until after 8AM. We measure the Time in Tool (G4), the number of forum questions, and the number of students that had completed the assignment on the first block of ten assignments. The students had to complete an assignment before they were able to continue to the next one. Two assignments (number six and nine), has two versions, each part of an experiment, which were randomly assigned to students. We determine correlation using Pearson's R and use Student's T to check for statistical difference between the two samples of students on each assignment. If there is a significant difference between these two groups, the time for students who start early cannot be used as predictor of time for students that start late.

3.2.6 Results

We compare the early students with those who complete the assignments after the first day. The average time measurements are presented in Table 4 as "early" and "late". There is a strong correlation (Pearson $R = 0.90$, $R^2 = 0.81$) between the early and late time averages. We also compare the early and late group using a two-tailed Students T-test to test for significant differences. We note that the first three assignments took longer for the students that started immediately to solve. Assignment 3 takes significantly longer time at $P < 0.05$. For the assignments after Assignment 5 we see that the students that started early do these assignments on average faster. A10 shows significantly different at $P < 0.01$.

3.2.7 Discussion

From Table 4 it seems reasonable that the eight students that completed all assignments early are also more skilled or motivated. Excluding Assignment 10 from the prediction model gives a very good fit (Pearson $R = 0.985$, $R^2 = 0.97$), but the significant difference in this assignment shows that judgments on how difficult an assignment is based on too few early measurements are unreliable.

The T-test results should be taken with a bit of skepticism since the later tasks are measured on a much smaller group of students. Both the normality

Table 4. The average times to solve assignments between early students and the students that completed the assignments after the first day (late). P is Student's-T p-values for two-tailed tests between early and late. * Significant at P < 0.05. ** Significant at P < 0.01

Assignment	**1**	**2**	**3**	**4**	**5**	**6.1**	**6.2**	**7**	**8**	**9.1**	**9.2**	**10**
Early (s)	239	245	1138	4078	4552	388	401	189	286	219	276	1775
Late (s)	190	153	737	5082	5188	530	926	520	685	648	699	5149
# Forum posts	0	0	0	4	4	1	0	0	0	0	1	2
Early N	38	37	36	33	25	8	3	8	8	3	5	8
P	0.23	0.17	*0.016	0.31	0.47	0.45	0.39	0.25	0.11	0.18	*0.032	**0.008

and the equal variance requirements on T-tests are violated and the measurements are not completely independent. When we do multiple T-tests, we might find significant P-values by pure chance.

Two assignments (4 and 5) stood out in terms of average time, number of compilations, and number of forum questions asked by the students. We based our conclusions on the faster students and missed the frustration of the struggling students in a third assignment (Assignment 10). We misunderstood the problems students faced on Assignment 4 and based our added lecture content on what we thought was the challenging part of the assignment. Inspections of Assignment 4 recordings would have revealed that students spent much time trying to understand the assignment and CSQUIZ itself. Helminen et al. [5] points out the problem of understanding a (new) tool. We used a group of six colleagues and two students to test CSQUIZ before the course started. While this group experienced some problems with interpreting CSQUIZ, most of the problems faced during the course were specific for the student group.

4. Limitations

The level of control of the programming situation is low; students most likely helped each other. We have seen at least one example of an unlikely API-method choice in several student solutions. We have not seen any indications that students have directly copied a solution into our tool, but cannot rule out that a student copied by writing. CSQUIZ fades text on inactivity; this most probably affected the "Active Use" granularity (G3) since students must stay active to continue reading. The data presented in this work is the result of a two classes of 66 and 83 undergraduate students. In the group of students, there is a subgroup of skilled programmers with years of experience, while others have little experience. This may have an effect on our prediction where a group of fast students completed the assignments early. We also present a limited sample of short PHP assignments, many of which included files ranging from a couple of lines to around 800 lines. The assignments are designed to be solved quickly (minutes to an hour) so our result may not be valid for longer sessions.

5. Related work

There are a number of tools that compute and use development time to support teaching and learning. ClockIt calculates and presents time to both students and teachers [3]. ClockIt capture start, exit, compile, save, error, and file change events using instrumentation in BlueJ. This is similar to our granularity level G2. Extrapolating from our results, assuming similar exercises and students, having a threshold of five minutes could give them acceptable time estimations (see Table 1). Rodrigo and Baker [4] also instruments on compilation level, and estimates student frustration. They intend to try more fine-grained approach with a full replay of keystrokes, mouse movements and other events. We think that having access to fine-grained recordings like ours would benefit their work.

Retina is another tool that collects compile and runtime errors [9]. Retina approximates time spent from the number of programming sessions (e.g., compilations that are more than thirty minutes apart). These approximations are used to present time predictions to students. This is an interesting approach and we intend to use our data to see if student solving time can be accurately predicted. Matsuzawa et al. [6] used detailed logging of key strokes in a tool to enable students to observe, analyze and improve their own programming process. They aggregate data on five minute thresholds and would thus (using our result) get a quite good estimate on average but get a high standard deviation. Ko et al. [10] let a group of ten developers solve five maintenance tasks during 70 minutes and captured their work using full screen recordings. They divided their activities to be even finer grained than ours. Using their results and aggregating similar activities show a combined reading and navigation time around 45% compared our "Active Use" of 37.9%. Their editing time was around 20% compared to our 14.8%. The main difference is that we compute these values automatically.

Marmoset[2] and Web-Cat[3] are popular assignment submission systems. Data from these systems (and modifications of them) have been used to find patterns in basic student behavior and high-level strategies used to solve the assignments. The major pattern found is that students that start early on an assignment typically perform better and students that submit late tend to continue to do so. Submitting late also correlates positively with lower grades [1]. More advanced analyses and model building have also been performed to find patterns in how students solve their assignments. For example, Hidden Markov Models was used to find problem-solving strategies used by students, where some strategies in the model correlate with higher student performance [11, 12]. This is in line with our results that early students may be faster on assignments (A4–A10). But we also see that for (A1 to A3) this group is actually slower in completing the assignments than the latter group.

[2] http://marmoset.cs.umd.edu
[3] http://web-cat.org

6. Conclusions and future work

We present an online tool, CSQUIZ, to help teachers understand how students solve programming assignments. Compared to similar tools we improved the granularity level of the recordings to include mouse movements, text-selections and text cursor positions. Increasing the granularity level requires more invasive instrumentation and a bigger effort. In this context we ask:

RQ1. How much of total tool usage time can be estimated using coarse-grained models? We compare different granularity levels and find that thresholds can be used on coarse grained models to get good estimations on average, but with large errors on individual measurements. Including mouse movement events reduces much of the error when estimating time spent in tool. The reason for the high errors on lower granularity levels is explained by the small relative proportion of editing time compared to reading time shown in Fig. 1.

RQ2. What is the fraction of assignment solving time spent on editing, reading and browsing assignment code and how much time do student spend outside of the tool? We are able to divide the time spent into editing, active use, passive use, and time out of tool. The amount of time for active use such as reading and navigating the tool is estimated to 37.9% on average. We calculate the maximum amount of time spent in the tool, but not actively interacting with it (on average 29.0%), thus providing a measurement of the time that cannot be accounted for but could be used to read code or instructions. We repeated the calculations for data from 2015 and found that on average the new group of students spends the same fractions of time on editing, reading, or out of tool.

RQ3. Can time measurements of students who complete assignments early predict what assignments the main group of students is going to struggle with? Descriptive statistics derived from fine-grained recordings further helped us identify which assignments were the most problematic for the students. To focus our teaching resources we compare students who submit early to students who submit late to find out if early submissions predict which assignments are going to be hard for the latter group. We see a correlation between early student submission assignment solving time and students that hand in later but also see an example where the times differ significantly, see Assignment 10 in Table 3. The eight early students that complete all ten assignments on the first day are also much faster than the latter students are. This group of fast students does not represent the main group of students.

We plan to use the fine-grained logging to examine how different assignments affect student behavior. We are aware that fine-grained logging (timed key strokes and mouse movements, etc.) is used in a different but related context; to understand the cognitive writing process [13], e.g., differences between novice and expert writers, and burst analysis (how much is written before pausing). In this context, logs have been complemented by eye-tracking and thinking-aloud protocols to help determine what the writer was looking at and thinking. Analysis using social network and data mining techniques has also been used to provide information about writing strategies using multiple electronic sources. It seems highly relevant to explore how students use online sources as part of their assignment solving process. We plan to explore these techniques in the future.

References

1. J. B. Fenwick Jr., C. Norris, F. E. Barry, J. Rountree, C. J. Spicer and S. D. Cheek, Another Look at the Behaviors of Novice Programmers. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, 2009, pp. 296–300.
2. M. C. Jadud, Methods and Tools for Exploring Novice Compilation Behaviour, *Proceedings of the 2nd Int. Workshop on Computing Education Research* (Canterbury, United Kingdom), 2006, pp. 73–84.
3. C. Norris, F. Barry, J. B. Fenwick Jr., K. Reid and J. Rountree, ClockIt: Collecting Quantitative Data on How Beginning Software Developers Really Work, *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (ACM, New York, NY, USA), 2008, pp. 37–41.
4. M. M. T. Rodrigo and R. S. Baker, Coarse-grained Detection of Student Frustration in an Introductory Programming Course, *Proceedings of the 5th Int. Workshop on Computing Education Research*, 2009, pp. 75–80.
5. J. Helminen, P. Ihantola, and V. Karavirta, Recording and Analyzing In-browser Programming Sessions, *Proceedings of the 13th Koli Calling Int. Conf. on Computing Education Research*, 2013, pp. 13–22.
6. Y. Matsuzawa, K. Okada and S. Sakai, Programming Process Visualizer: A Proposal of the Tool for Students to Observe Their Programming Process, *Proc. of the 18th ACM Conf. on Innovation and Technology in Computer Science Education*, 2013, pp. 46–51.
7. A. Vihavainen, J. Helminen and P. Ihantola, How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces, *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (ACM, New York, NY, USA), 2014, pp. 109–116.
8. I. Utting, N. Brown, M. Kölling, D. McCall and P. Stevens, Web-scale Data Gathering with BlueJ, *Proceedings of the 9th Int. Conf. on Computing Education Research*, 2012, pp. 1–4.
9. C. Murphy, G. Kaiser, K. Loveland and S. Hasan, Helping Students and Instructors Based on Observed Programming Activities, *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (ACM, New York, NY, USA), 2009, pp. 178–182.
10. A. J. Ko, B. A. Myers, M. J. Coblenz and H. H. Aung, An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, *IEEE Trans. Softw. Eng.*, **32**, 2006, pp. 971–987.
11. C. Piech, M. Sahami, D. Koller, S. Cooper and P. Blikstein, Modeling How Students Learn to Program, *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 2012, pp. 153–160.

12. U. Kiesmueller, S. Sossalla, T. Brinda and K. Riedhammer, Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods, *Proceedings of the 15th Conf. on Innovation and Technology in Computer Science Education*, 2010, pp. 274–278.
13. M. Leijten and L. Van Waes, Keystroke Logging in Writing Research, *Written Communication*, **30**, 2013, pp. 358–392.

Daniel Toll received a M.A. degree in Computer Science in 2004 from Växjö University. He is a lecturer in Computer Science at Linnaeus University. Daniel is also a Ph.D. student with primary research in learning analytics, educational data mining, and computer science education.

Tobias Olsson received a B.Sc. degree in Software Engineering in 2000 from Blekinge Institute of Technology. He is a lecturer in Computer Science at Linnaeus University. Tobias is also a Ph.D. student with primary research in software quality and software architecture conformance checking.

Morgan Ericsson received a B.Sc. degree in Computer Science in 2001 and a Ph.D. degree in Computer Science in 2008, both from Växjö University (Växjö, Sweden). He is currently a senior lecturer in Computer Science at Linnaeus University (Växjö and Kalmar, Sweden). Dr. Ericsson's primary research interest is software and information quality assessment, network analysis, data mining, and software analytics.

Anna Wingkvist received her Ph.D. degree in Computer Science with specialization in Information Systems in 2009 from Växjö University (Växjö, Sweden). She is currently a senior lecturer and a research advisor at Linnaeus University (Växjö and Kalmar, Sweden). Dr. Wingkvist's current research interest is methodology, methods, and tool reasoning for software and information quality assessment.